

Efficient Booleans algorithms for triangulated meshes of geometric modeling

Xiaotong Jiang¹ , Qingjin Peng² , Xiaosheng Cheng¹ , Ning Dai¹ , Cheng Cheng¹  and Dawei Li¹ 

¹Nanjing University of Aeronautics and Astronautics, China; ²University of Manitoba, Canada

ABSTRACT

Boolean operation of geometric models is an essential element in computational geometry. An efficient approach is developed in this research to perform Boolean operation for triangulated meshes represented by B-rep. This approach is much fast and robust than many existing methods. The Octree technique is adapted to facilitate the division of the common space of two meshes in order to reduce the time of Octree's construction and intersection detection. Floating point arithmetic errors and singularity of intersections are then analyzed to guarantee the unique intersection between a segment and a face, and the continuity of intersections. A novel technique based on intersecting triangles is finally proposed to create required sub-meshes based on the type of Boolean operations. Some experimental results and comparisons with other methods are presented to prove that the proposed method is fast and robust.

KEYWORDS

Octree; floating point arithmetic errors; singularity of intersections; sub-meshes classify; fast and robust

1. Introduction

Boolean operations of geometric models are important processes in computational geometry [7,15,16]. A most commonly used method in Boolean operations to display geometric models is the boundary representation (B-Rep) based on parametric surfaces. Booleans operations on B-Rep models were introduced in the 1980s [24, 30]. With the development of reverse engineering and 3D printing technologies, the use of discrete surfaces, especially the triangulated meshes, is increasing. Triangulated models are used in many fields, such as architectural design, industrial design, and CAE. Numerous Boolean algorithms have been proposed to resolve triangulated meshes in geometric modeling. However, the complex problem challenges performance of these methods in efficiency and robustness.

Boolean algorithms can be classified into three broad categories: exact arithmetic, approximate arithmetic and volumetric methods [6]. The exact arithmetic operates Boolean operations directly over geometric elements, such as faces, edges and vertices [2, 30]. However, they often suffer the problem in searching stable solutions. Various methods have been proposed to improve the robustness of the exact arithmetic. Boolean operations were implemented by Keyser [21] in a curved domain for an exact arithmetic. An interval computation is adopted by Hu [17, 18] to operate Booleans on solid models with spline surfaces. The BSP (binary space partition) technique was proposed by Bernstein and Campen [3, 4] to

improve the stability of the exact arithmetic. To avoid the complex exact computation and effectively deal with degenerated intersections, approximate arithmetic methods and volumetric methods were suggested. For approximate arithmetic methods, Smith and Dodgson [32] presented a topologically robust algorithm. Ming et al. [6] proposed an approach to compute the approximate Boolean operations of two free-form polygonal solids efficiently with the help of Layered Depth Images. The volumetric method converts mesh surfaces into the volumetric representation. Booleans can then be operated based on volumetric data [10, 27]. Because of the limited precision of the representation, the loss of geometric features is unavoidable. Marching Cubes algorithm is usually used to extract features from the volumetric result [22]. Pavic et al. [29] presented a hybrid method to combine the volumetric technique with surface-oriented techniques. The method can obtain an output mesh to preserve the existing sharp features and reconstruct new features appearing along intersections of the input meshes. This work was for the exact Booleans using B-Rep triangle meshes which are exactly edge-based data structures [25].

Robustness and efficiency of exact Booleans on triangulated meshes have proven a challenging and complicated task. There are two main procedures of the triangulated meshes in Boolean algorithms: the intersection detection between two meshes and Boolean operations. Intersection detection and Boolean operations are crucial

to the performance of the algorithm. This paper presents a fast and stable intersection detection technique and Boolean operations to achieve excellent performance.

For the intersection detection between two meshes, many techniques have been proposed to accelerate the speed of intersection detections. A BSP-based method was used to accelerate the spatial queries in the process [3]. The method is stable and can operate complex meshes, however it encounters memory issues when meshes have a large number of faces [9]. A technique was adopted by Campen [4] to combine an adaptive Octree with the nested binary space partition, which can achieve the high performance and less memory consumption. Jing et al. [20] adopted an oriented bounding box (OBB) tree method. The OBB tree is fast to obtain the intersection lines once it is built. However, the building time of the structure is slow and is inconvenient to use. Feito et al. [9] used the Octree method which requires the less storage and can run in a multithreaded environment. The Octree method has the good performance to deal with complex meshes. Some methods were proposed to calculate the intersection line of a pair of triangles. Moller [26] presented a method to compute whether or not two triangles intersect. Tropp et al. [35] developed an optimum method to save about 20% time of the mathematical operations. When calculating intersection lines of two meshes, there are two factors that influence efficiency and stability of the algorithm. The first is that each intersected edge of one mesh will produce two points with another mesh. The two points have the same coordinate value theoretically. The floating point arithmetic errors may cause the discontinuity of intersections. The second is that there may be singularity when judging the intersection

between a segment and a mesh. This also influences the continuity of intersections.

For Boolean operations, a key technique is the point-in-solid test to classify required sub-meshes. Several approaches can be used to perform the point-in-solid test. Feito et al. [28] used the simplicial covering of the solids. This method is very fast when it is implemented on GPU, however it needs a large number of tests to classify the newly created sub-mesh. A method of Jordan Curve Theorem was adopted by Veblen [36], the method uses a ray-surface intersection test in order to obtain the parity of the number of intersections. The method is fast, however it needs the Octree calculated in advance, the method may fail when it deals with the open mesh. Guo et al. [13] used a loop detection to decide the candidate point whether inside or outside the solid. However, these methods are only designed for solid models, and require meshes no boundary edges. Jacobson et al. [19] used a generalized winding numbers method for the inside-outside segmentation to handle the open mesh situation. However, there is still not a unified rule of Booleans for open meshes. As shown in Fig. 1, using the commercial CAD software Geomagic Studio 12, the Booleans difference operation is applied on two open meshes. If an open mesh's normal is flipped, the result will be difference.

2. Method

In this paper, a fast and robust Boolean algorithm for the triangulated mesh is presented. It involves two states as shown in Fig. 2: intersection detection between two meshes and Boolean operations. For the intersection detection between two meshes, the Octree technique is

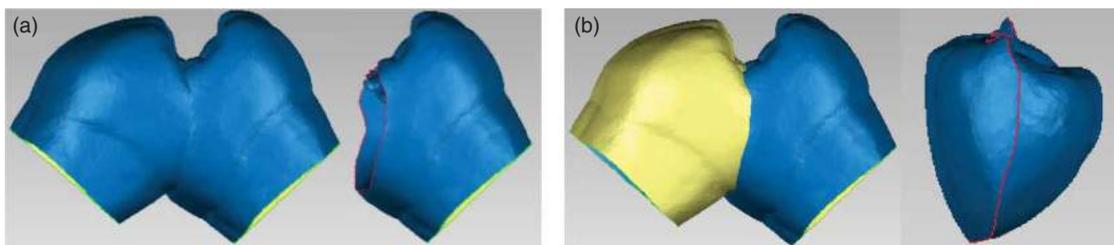


Figure 1. Booleans difference results using Geomagic Studio 12.

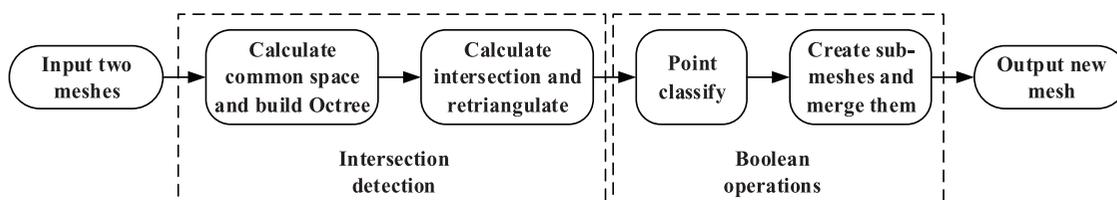


Figure 2. Flow of the approach.

adapted to facilitate the division of the common space of two meshes in order to reduce the time of Octree's construction and intersection detection, floating point arithmetic errors and singularity of intersections are analyzed to find intersections. In this way, the unique intersection between a segment and a mesh can be obtained to guarantee the continuity of intersections. For Boolean operations, a fast and stable technique is presented to realize union, intersection and difference operations. The method is simple as it finds only points that belong to these intersected triangles to create required sub-meshes based on the type of Booleans. The method is suitable for both closed and open meshes.

In summary, the main contributions of this work are as follows:

- (1) Octree is used to facilitate the division of the common space of two meshes in order to accelerate the speed of intersection detection and reduce memory utilization.
- (2) Floating point arithmetic errors and singularity of intersections are analyzed to improve the stability of the algorithm.
- (3) A stable technique based on intersected triangles is presented to realize union, intersection and difference operations. The method is fast for both closed and open meshes.
- (4) The algorithm is robust and can be used in a similar milling simulation system which contains lots of Boolean difference operations between a workpiece and a cutter.

2.1. Intersection detection

Intersection detection between two meshes is the basis of Boolean operations. Searching for intersection lines fast and exactly is a key to successful Booleans. In this section, a memory-saving method is introduced to accelerate the speed of intersection detection. The floating point arithmetic errors and singularity of intersections are also analyzed for the intersection test to obtain continuous intersections.

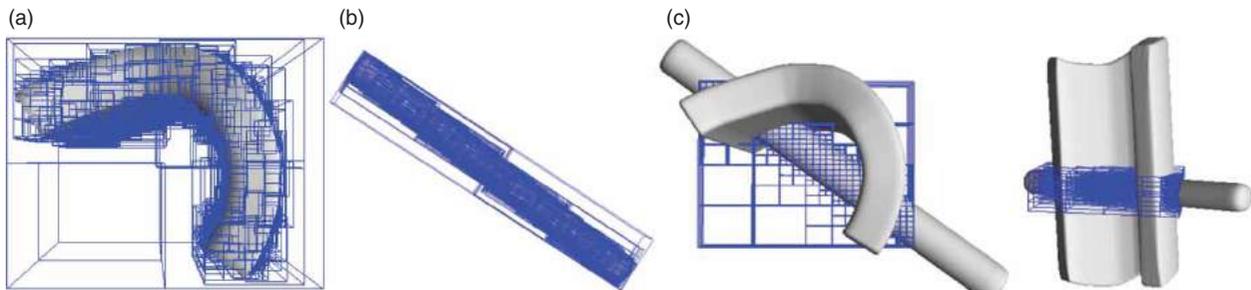


Figure 3. Octree used in meshes intersection. (a) & (b) Octree dividing whole meshes. (c) Octree dividing the common space of two meshes.

2.1.1. Building Octree of the common space

Many techniques have been used to accelerate the intersection detection of two meshes, such as methods of BSP (binary space partitions) [3, 4], OBB tree [20], Spatial hashing [8, 33, 38], and Octree [9]. The Octree technique is adopted in this research.

For two intersected meshes, the size of the common space is small. Octree can be built to divide the common space to accelerate the speed of intersection detection and reduce memory utilization. The common space can be calculated as follows:

Given two triangulated meshes S_A and S_B as shown in Fig. 3. Let Box_A and Box_B be the smallest axis-aligned bounding boxes (AABBs) of S_A and S_B . Box_A and Box_B shown in Fig. 3 (a) and (b) can be written as follows:

$$Box_A = \begin{pmatrix} x_{Amax}, y_{Amax}, z_{Amax} \\ x_{Amin}, y_{Amin}, z_{Amin} \end{pmatrix},$$

$$Box_B = \begin{pmatrix} x_{Bmax}, y_{Bmax}, z_{Bmax} \\ x_{Bmin}, y_{Bmin}, z_{Bmin} \end{pmatrix}.$$

The common space $Box_A \cap^B Box_B$ can be calculated as:

$$Box_A \cap^B Box_B = \begin{pmatrix} \min(x_{Amax}, x_{Bmax}), \min(y_{Amax}, y_{Bmax}), \\ \max(x_{Amin}, x_{Bmin}), \max(y_{Amin}, y_{Bmin}), \\ \min(z_{Amax}, z_{Bmax}), \\ \max(z_{Amin}, z_{Bmin}) \end{pmatrix}$$

The AABBs is expanded to ensure the common space containing all the intersected triangles. The modified result is as follows:

$$\overline{Box_A \cap Box_B} = \begin{pmatrix} \min(x_{Amax}, x_{Bmax}) + l, \min(y_{Amax}, y_{Bmax}) + l, \\ \max(x_{Amin}, x_{Bmin}) - l, \max(y_{Amin}, y_{Bmin}) - l, \\ \min(z_{Amax}, z_{Bmax}) + l, \\ \max(z_{Amin}, z_{Bmin}) - l \end{pmatrix}$$

where l is the longest edge of S_A and S_B .

Table 1 shows the intersection detection time of two different building methods of Octree. The two building

Table 1. Time comparison of intersection detections.

Triangles of meshes (A+B)	Octree is built to divide whole meshes(s)	Octree is built to divide the common space of meshes (s)
345944+345944	6.205	4.428
345944+276754	5.656	4.042
345944+221402	5.308	3.735
345944+177120	5.063	3.157
345944+106272	4.537	2.829

methods are (A) Octree built to divide whole meshes, and (B) Octree built to divide the common space of meshes.

The Octree can be configured to achieve a good balance between the building time and the performance of queries on spatial data. The memory usage of Octree is also affected by this configuration. Building the Octree usually needs $O(N \log(N))$ for a set of N triangles. Building Octree starts with the common minimum AABB of two meshes. Each node is divided into eight sub-nodes until all nodes meet the termination condition. The termination condition is decided by two parameters: the maximum number of triangles per leaf node and the maximum depth of the Octree. In order to avoid excessive consumption of memory in extreme cases (for example, there are many overlapped triangles in a mesh), the depth of the Octree has a higher priority. Generally speaking, an Octree with a depth of 8–10 and a limit of about 50 triangles per node has the nearly optimal performance [9]. The Octree in this research is configured based on these values of the depth and triangles per node.

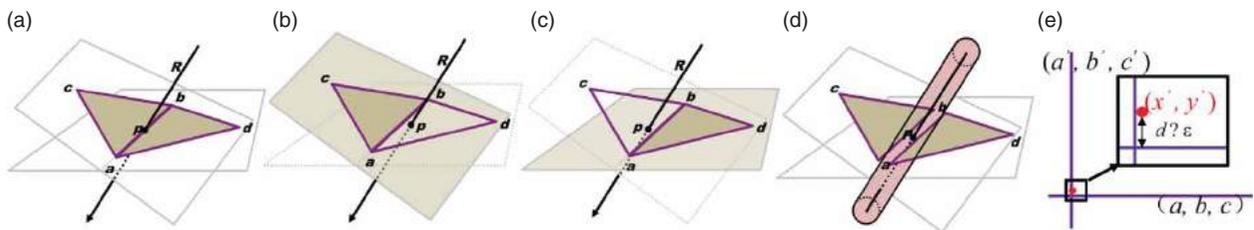
2.1.2. Floating point arithmetic errors and singularity of intersections

Floating point arithmetic is inevitable in computing geometry. Especially in Boolean operations, there are many intersections for computation. Floating point arithmetic may cause discontinuity of intersections, which further causes failure of Booleans.

Intersections computation in Booleans can be regarded as many Ray-triangle tests. Intersecting a ray against a triangle is far more problematic than common “solutions” assumed. Fig. 4 shows how an incorrect computation can

lead to catastrophic failures. It is also shown how to perform the calculation robustly. The reason is that this test is not robust, it fails for intersection against two triangles sharing an edge. As shown in Fig. 4(a), a ray, R , strikes the shared edge ab of $\triangle abc$ and $\triangle abd$. First the intersection point p of the ray R with the plane of $\triangle abc$ is computed. Let's say small errors in the computation of p put p slightly to the right of ab such that the containment test for p being inside $\triangle abc$ fails, See Fig. 4(b). Then the intersection point p of the ray R with the plane of $\triangle abd$ is also computed. This computation uses completely different floating-point values than the previous computation, we can get a point p slightly different from the previous one. Let's say this has small errors in the computation of p to put p slightly to the left of ab such that the containment test for p being inside $\triangle abd$ fails, See Fig. 4(c). R misses both triangles, seemingly passing through the shared edge. When it comes to this, the intersection is broken at the points of p . An alternative to make a ray test robust is to switch to a “fat” query primitive, See Fig. 4(d). For example, here a sphere is swept against the triangles. Fat tests allow intersections to be found even if the triangles have a gap between them, as long as the width of the gap is smaller than the “fatness diameter” of the query object.

The fat test is actually a tolerance value problem. A line can be defined as a set of points $\{(x, y) \in \mathbb{R}^2: ax + by + c = 0\}$. Let $x' = (bc' - b'c)/(ab' - a'b)$ and $y' = (ca' - b'c)/(ab' - a'b)$ are the coordinates of the intersection point between two non-parallel lines: (a, b, c) and (a', b', c') , as shown in Fig 4(e). By definition, point (x', y') lies on line (a, b, c) if $ax' + by' + c = 0$. Suppose that there is a simple, floating point implementation that determines the point of intersection according to the above formula. When $ax' + by' + c$ is computed to be exactly 0, point (x', y') lies on line (a, b, c) . Because of approximation errors, it often fails to judge the intersection point between two lines to lie on one or other line [23]. A common method to solve this issue is to apply arithmetic tests within a given tolerance. Hence, the point-on-line test is operated by determining whether the value $ax + by + c$ is less than some selected tolerance value, $\varepsilon > 0$. If the line representations are always held

**Figure 4.** Ray-triangle test.

in $a^2 + b^2 = 1$ form, then $ax' + by' + c$ represents the signed distance from the line. If $ax' + by' + c = d \leq \varepsilon$, point (x', y') lies on line (a, b, c) , otherwise not, as shown in Fig. 4 (e).

Singularity of intersections also can break the intersection. Suppose there are two polyhedras used to calculate the intersection line, see Fig. 5. Firstly, intersection line loops of the two meshes are calculated. These loops are from face to face intersections. Second, these intersection surfaces of the two polyhedral objects are partitioned into new surfaces. The process of this partition relies heavily on traversals of adjacency intersections. When doing intersection detection between two meshes, we locate the intersection points on edges of one mesh with faces of another mesh that is where a face/face intersection ends.

Singularity of intersections often results from different angles between edges and faces. In Fig. 5(a), it is assumed that the front edge of the tetrahedron intersects the top face of the cube at a steep angle and the front face of the cube at a shallow angle. The two faces share the common edge ef . When carrying out the intersection detection, firstly, the intersection point between the front edge and the top face is calculated. Let m is the intersection point, see Fig. 5 (b) & (c), d_1 is the distance of m to the edge ef . As the steep angle, it is possible that the intersection point between the edge and the top face is deemed to lie on edge of the face according to the fat test: $ax_A + by_A + c = d_1 < \varepsilon$. Secondly, the intersection point between the front edge and the front face is calculated. Let m^* is the intersection point, see Fig. 5(b) & (c),

d_2 is the distance of m^* to the edge ef . As a shallow angle, it is obvious that $d_2 > d_1$. It is therefore possible that the intersection point between the edge and the front face is deemed to lie inside the front face according to the fat test: $ax_B + by_B + c = d_2 > \varepsilon$. The front edge of the tetrahedron now intersects the front face twice, at m and m^* . It is logically inconsistent and the singularity arises. This will break the intersection loops and the algorithm would fail as a consequence.

2.1.3. Calculating intersection lines

When implementing intersection lines calculation, it needs to consider two triangles that are coplanar or not. In the coplanar case, it can be reduced to a 2D intersection problem for the efficient calculation. This paper mainly discusses the non-coplanar case. In section 2.2, floating point arithmetic errors and singularity of intersections were discussed, which could break the intersection. An improved intersection detection strategy is introduced to guarantee the continuity of intersections. This contributes the robustness of our algorithm.

Generally speaking, there are three intersection types between edge and triangle as shown in Fig. 6: the point of intersection is one endpoint of an edge, the point of intersection lies on one edge of triangle, and the point of intersection lies inside the triangle. Property of these three types are set as EndP, EdgeP and FaceP, respectively. Among the three types, the EndP has the highest weight, the EdgeP has the moderate weight and the FaceP has the lowest weight. In the previous paragraph, intersection

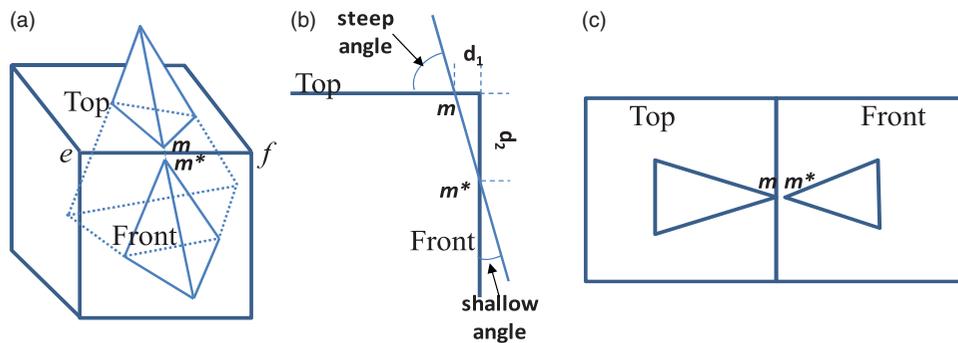


Figure 5. Singularity of intersections.

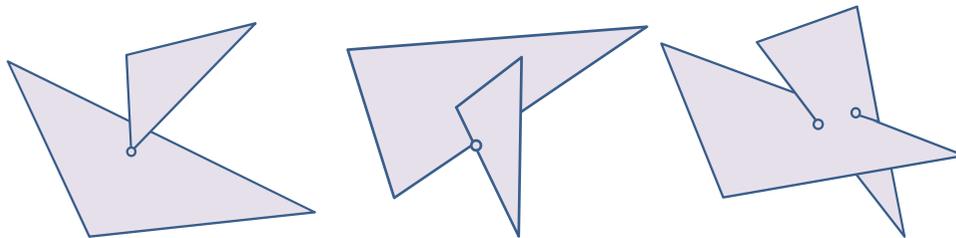


Figure 6. Intersection of edge and triangle.

points m and m^* are obtained. According to the type of intersection, property of m and m^* are set as EdgeP and FaceP, respectively. Because m has a higher weight than m^* , the property and coordinate of m^* is reset as the same as m . If m and m^* have the same property, the property and coordinate of m^* is just reset as the same as m . In this case, an edge has unique intersection point with a mesh at the same location. Details of intersection line calculation are listed in Algorithm 1.

Algorithm 1 Calculate intersection lines

```

for each pair of triangles ( $T_1, T_2$ ) do
  for each edge  $e \in T_1$  do
     $m = \text{Intersection}(e, T_2)$ ;
    if ( $\text{exist\_intersection}(e, T_2)$  &&
       $m^* = \text{Intersection}(e, T_2)$ )
      Properties( $m$ ) = Properties( $m^*$ ) =
      Priority( $m, m^*$ );
      Coordinate( $m^*$ ) = Coordinate( $m$ );
    end for
  for each edge  $e \in T_2$  do
     $m = \text{Intersection}(e, T_1)$ ;
    if ( $\text{exist\_intersection}(e, T_1)$  &&
       $m^* = \text{Intersection}(e, T_1)$ )
      Properties( $m$ ) = Properties( $m^*$ ) =
      Priority( $m, m^*$ );
      Coordinate( $m^*$ ) = Coordinate( $m$ );
    end for
  end for

```

2.1.4. Retriangulating intersected triangles

After calculating the intersection for all triangle-pairs, for an individual intersected triangle, usually there are several intersection lines inside the triangle since it perhaps intersects with several other triangles, and it will be divided into several polygons by these intersection lines. All resulting polygonal faces should be decomposed into new triangles and the topology of the mesh is updated. For the polygon triangulation, there are at least three popular algorithms: Recursive Ear Cutting algorithm improved by Held [14] and Toussaint [34], Incremental Randomized Algorithm proposed by Sediál [31] and Sweep Line algorithm presented by Garey et al. [11]. Recursive ear cutting algorithm is easy to implement compared to other complicated algorithms. However, the algorithm has the poor performance as it is difficult to extend it to polygon with holes. On the other hand, the incremental randomized algorithm has better performance, however, the improved algorithm presented by Nancy et al. [1] is very difficult to implement. The sweep line algorithm is the most widely used

algorithm in nowadays real applications. Wu [37] proposed an optimized sweep line algorithm called Poly2Tri to handle the polygon with holes. Poly2Tri is adopted in this paper. The result is shown in Fig. 7.

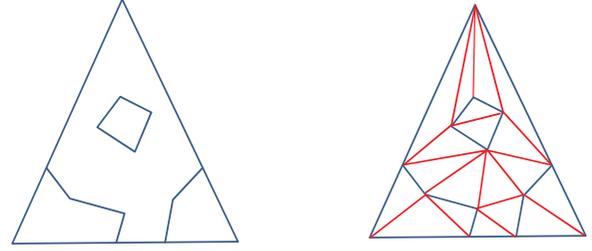


Figure 7. Polygon triangulation of Poly2Tri algorithm.

2.2. Boolean operations

In this section, details of Boolean operations on triangulated meshes are delivered. Our method has two steps: to identify if a point is inside or outside a mesh, and to create sub-meshes and merge them. This paper proposes a technique to only identify points that belong to these intersected triangles. It only utilizes these points to create required sub-meshes based on the type of Booleans. This method is fast and can be used to both closed and open meshes.

2.2.1. Point classify

In almost all literature, a ray-surface intersection method is used to identify if a point is inside or outside a mesh. However, this method is designed based on the assumption that a mesh is a solid without boundaries. When there is an open mesh with boundaries, the method often fails, as there is no strict inside and outside for open meshes. It is difficult to identify the location of the point. This paper proposes a simple method to decide point locations for open meshes of Booleans. The method classifies only points that belong to these intersected triangles.

Let mn is the intersection line of Δabc and Δefg . The two triangles belong to two meshes, respectively. n_{abc} and n_{def} are the normal of Δabc and Δefg . The directions of normal are (x_{n1}, y_{n1}, z_{n1}) and (x_{n2}, y_{n2}, z_{n2}) respectively. $p(x_1, y_1, z_1)$ and $q(x_2, y_2, z_2)$ are two arbitrary points that lie in planes of Δabc and Δefg respectively. Equations of the two planes are as follows:

$$F_{abc}(x, y, z) = (x - x_1).x_{n1} + (y - y_1).y_{n1} + (z - z_1).z_{n1} = 0 \quad (1)$$

$$F_{def}(x, y, z) = (x - x_2).x_{n2} + (y - y_2).y_{n2} + (z - z_2).z_{n2} = 0 \quad (2)$$

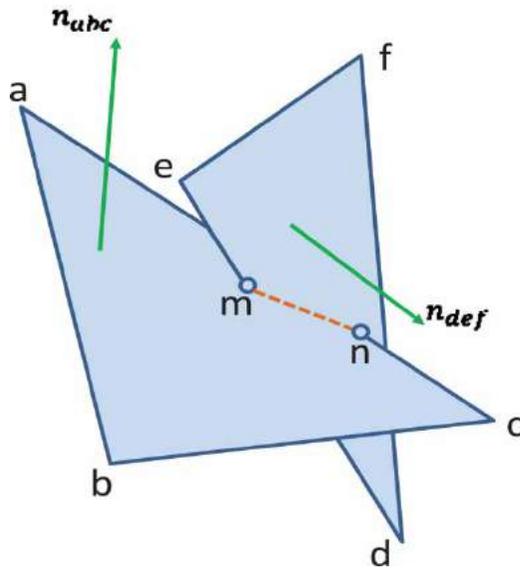


Figure 8. Point classification.

As shown in Fig. 8, the edge ac and Δdef have an intersection point n . For the vertex $a(x_a, y_a, z_a)$, $F_{def}(x_a, y_a, z_a) < 0$. a lies inside of the mesh which contains Δdef . Following the same way, d lies inside, c and e lie outside. In a degenerated intersection case, there is at least a vertex $v(x, y, z)$ of one triangle that lies on another mesh. For $v(x, y, z)$, $F(x, y, z) = 0$ and the vertex is labeled “on”. When all vertexes that belong to the intersected triangles have been judged, vertexes lie inside a mesh are labeled “in”, vertexes lie outside the mesh are labeled “out” and these intersection points are labeled “on”. The following creating sub-meshes will depend on these labeled vertexes.

2.2.2. Creating sub-meshes and merging them

There are three types of Boolean operations: union, intersection and differences. If there are two meshes M_A and M_B , and M_{AinB} is a set of triangles from M_A inside M_B . M_{AoutB} is a set of triangles from M_A outside M_B . M_{BinA}

is a set of triangles from M_B inside M_A . M_{BoutA} is a set of triangles from M_B outside M_A . M_{onAB} is a set of triangles from Both M_A and M_B (when there are coplanar intersections). The three types Boolean operations can be presented as follows ($M_A - M_B$):

$$M_A \cup^{M_B} (\text{union}): M_{AoutB} + M_{BoutA} + M_{onAB}$$

$$M_A \cap^{M_B} (\text{intersection}): M_{AinB} + M_{BinA} + M_{onAB}$$

$$M_A - M_B (\text{difference}): M_{AoutB} + M_{BinA}$$

In this paper, the method only utilizes above labeled vertexes to create required sub-meshes. For instance, the process is as follows if there is a need to create sub-meshes inside a mesh. Firstly, selecting a vertex V_{in} labeled “in”, see Fig. 9(a), shown as the red vertex. Secondly, the selected vertex grows according to the topology of the mesh with these intersection lines as borders. The procedure terminates when the number of vertex doesn't increases, as shown in Fig. 9(b). Thirdly, vertexes labeled “in” that haven't been visited are handled using the same method. When all vertexes labeled “in” are visited, the required sub-mesh is created as shown in Fig. 9(c). Details of creating sub-meshes are listed in Algorithm 2. After all required sub-meshes are created, these meshes are merged to complete Booleans. Fig. 10 shows some samples of Booleans using our method. Fig. 10(a) is Booleans between a bunny model (closed mesh) and a Schwarz P-surface (open mesh). Fig. 10(b) is Booleans between two sphere models, there are many singular intersections. Fig. 10(c) is Booleans between two armadillo models. The two models are complex with 350000 triangles.

3. Results and discussion

The algorithm code has been completely written using C++. Our Booleans algorithm is tested using some triangulated meshes. Some other systems are also tested

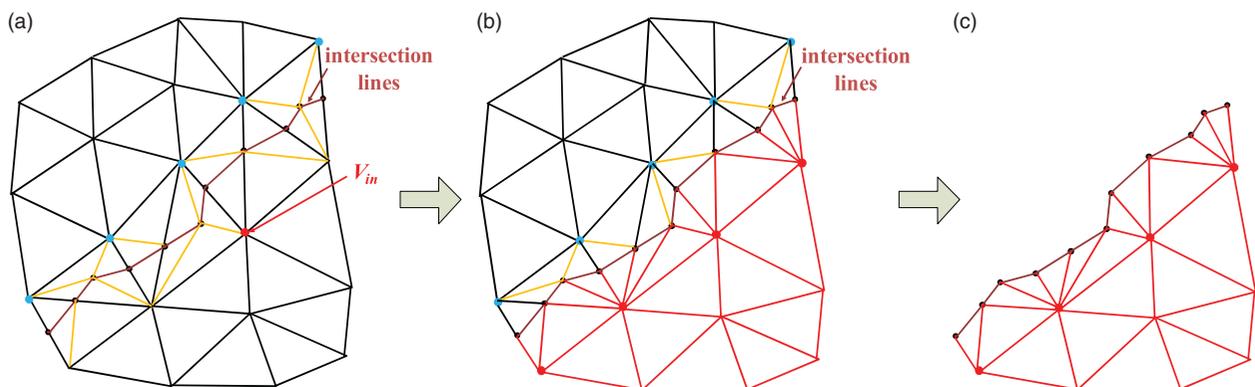


Figure 9. Creating required sub-meshes.

Algorithm 2 creating required sub-meshes (such as M_{AinB})

```

 $M_{new}$ : a new empty mesh
 $V_{new}^*$ : the set of intersected points
for each  $V$  labeled “in” &&  $V \notin M_{new}$  do
  add  $V$  into a contain  $Ver\_C$ ;
  for each  $V_c \in Ver\_C$  do
    search all neighbouring vertexes  $V_{nei-C}$  and triangles  $T_{nei-C}$  of  $V_c$ ;
    for each  $V' \in V_{nei-C}$  &&  $V' \notin V_{new}^*$  &&  $V' \notin M_{new}$  do
      add  $V'$  into  $M_{new}$ ;
      add  $V'$  into  $Ver\_C$ ;
    end for
    for each  $T' \in T_{nei-C}$  &&  $T' \notin M_{new}$  do
      add  $T'$  into  $M_{new}$ ;
    end for
  remove  $V_c$  from  $Ver\_C$ ;
  end for
end for

```

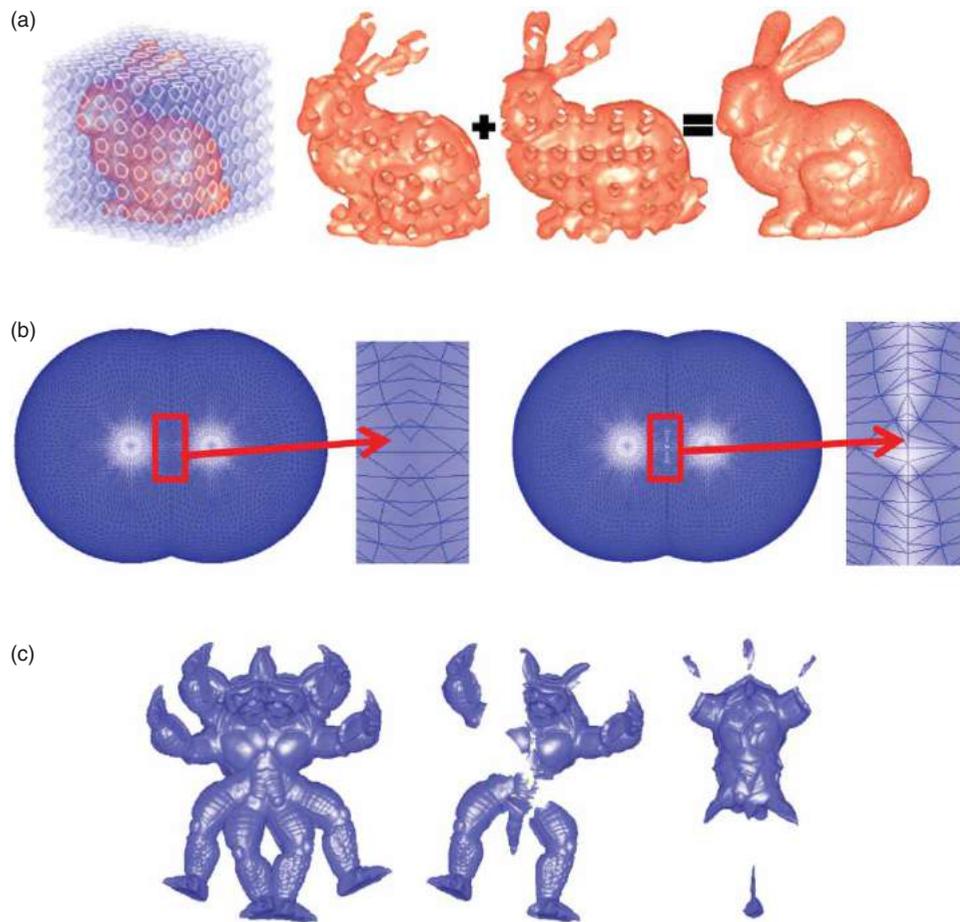


Figure 10. Samples of Booleans (a) bunny and Schwarz P-surface (closed and open mesh) (b) two sphere models (singular intersections) (c) armadillo models (complex model).

in order to compare the quality of the results and the overall performance including CGAL [5], the GUN triangulated surface library (GTS) [12], Rapidform 2006 and Gilbert Bernstein et al.’s method [3]. All of these

tests have been carried out on a computer with 3.00 GHz Inter Core2 Duo CPU and 4GB RAM. Table 2 shows the performance time of these tests. Each result is calculated as the average of time for all three Boolean

operations: union, intersection and differences of two same meshes which have a different position in space. A variety of results have been obtained. Based on the comparison of results, our algorithm shows the best performance using the least time. Gilbert et al.'s method, GTS and Rapidform 2006 are relatively slow, CGAL is the slowest. When the models are complex, the advantage of our algorithm is obvious. Our method, Rapidform 2006 and Gilbert et al.'s method can handle all the five meshes. In order to further validate the stability of our algorithm, the algorithm is also tested in a milling simulation system. The process of milling simulation is actually the removal of material from a workpiece, which contains abundant Boolean difference operations between the workpiece and a cutter. Most of the milling simulation systems are developed based on 2D environments or based on Multi-Resolution 3D Models. However, it is difficult to use Boolean algorithm to simulate the process because the results of Boolean operations often contain many self-intersections, overlapping and non-manifold triangles which greatly reduce the stability of

Boolean algorithms. It is challenging to implement consecutive Boolean operations for many existing Boolean algorithms. Our method successfully uses a cube model and a cylindrical ball-head model to imitate three processes of machining in a real 3D environment. Fig. 11 shows consecutive Boolean operations in a milling simulation system. A cylindrical ball-head milling cutter cuts a cube model using the Boolean difference operation. The leftmost is the original model. The remaining three pictures are results of the three processes of milling simulation containing 1200 Boolean difference operations. The bottom pictures are top views of the corresponding mesh, respectively.

Table 2 lists the performance comparison of different systems in Boolean operations. Each result is calculated for the average of time for all Boolean operations. The time of our algorithm contains the whole processing pipeline of the Booleans, including all required conversions. The Iphigenie model is from the reference [4], which provides the materials on their project homepage. Table 3 shows the performance time of different types of

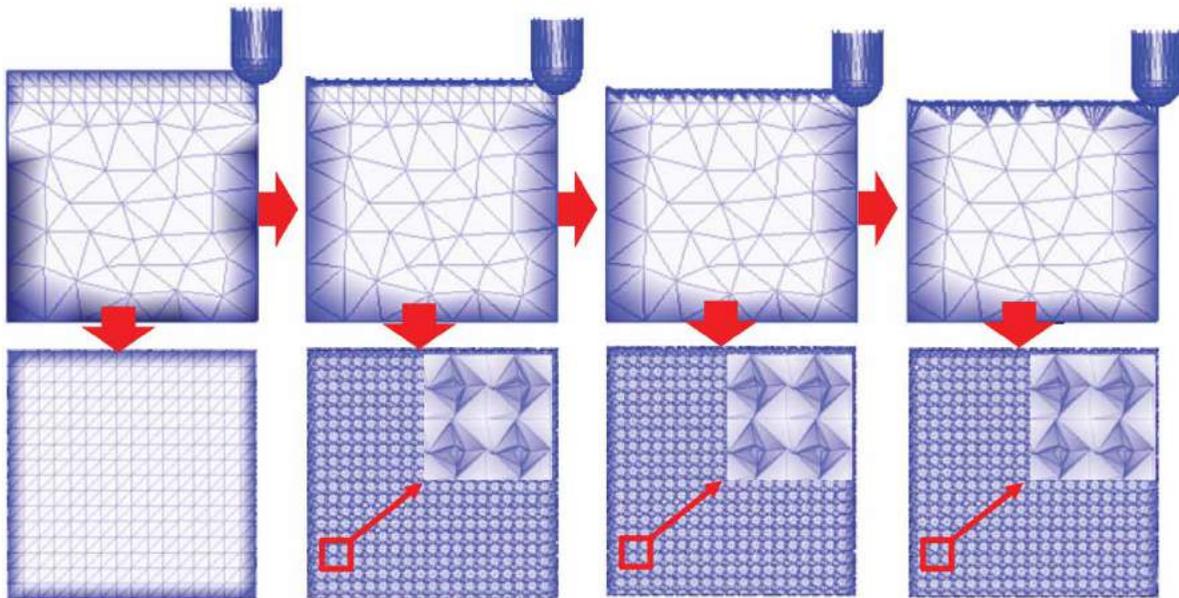


Figure 11. Boolean operations for the milling simulation.

Table 2. Performance comparison of different systems in Boolean operations.

Model(Triangle Number)					
Methods	Sphere (8.448 K)	Bunny (71.69 K)	Horse (158.7 K)	Armadillo (345.9 K)	Iphigenie (800 K)
CGAL	10.023s	80.45s	Failed	Failed	Failed
GTS	0.255s	2.015s	Failed	7.505s	Failed
Rapidform 2006	Less than 1s	≈ 2.05	≈ 6.250s	≈ 20.15s	≈ 101.0s
Gilbert et al.'s method	0.177s	1.878s	4.221s	9.952s	26.423s
Our method	0.179s	1.859s	3.112s	6.068s	14.752s

Table 3. Performance time (Intersection detection/Boolean operation/total time) of different types of Boolean operations (Union, Difference, Intersection) for different models.

Model(Triangle Number)	 Sphere (8.448 K)	 Bunny (71.69 K)	 Horse (158.7 K)	 Armadillo (345.9 K)	 Iphigenie (800 K)
Boolean operation type					
Union	0.089s/ 0.090s/ 0.179s	0.862s/ 0.927s/ 1.789s	1.505s/ 1.476s/ 2.981s	3.014s/ 2.993s/ 6.007s	7.502s/ 6.900s/ 14.402s
Difference	0.088s/ 0.088s/ 0.176s	0.859s/ 1.007s/ 1.866s	1.598s/ 1.529s/ 3.127s	3.102s/ 3.036s/ 6.138s	7.512s/ 7.115s/ 14.627s
Intersection	0.090s/ 0.091s/ 0.181s	0.869s/ 1.028s/ 1.897s	1.587s/ 1.727s/ 3.314s	3.104s/ 3.014s/ 6.118s	7.511s/ 7.492s/ 15.003s

Boolean operations (Union, Difference, Intersection) for different models.

CGAL and GTS are two geometric processing libraries which provide Boolean algorithms. CGAL provides tools for processing polygons. It is designed to allow arbitrary precision arithmetic Booleans. The performance of CGAL is acceptable for low-complex meshes. However, when meshes are complex such as more than 200 K triangles, it becomes slow and may even fail, because the internal representation already consumes about 5.3GB of the main memory [4]. On the other hand, GTS has good performance in the processing speed as it uses an AABB tree to accelerate the process of intersection. GTS can also process complex meshes and the quality of results is high. However, it is very strict for the topology of meshes. Any self-intersecting, inconsistency or nonorientable of input meshes, such as the Horse and Iphigenie, will cause the failure of the Booleans. It is therefore weak in implementing consecutive Boolean operations. It successfully completed only 10 Booleans operations in imitating the process of the milling machining example.

The commercial packages of Rapidform 2006 have good performance to produce correct results for all the tested meshes. It is very fast when the model is low-complex (less than 200 K triangles). However, when the input mesh is large, the time increases sharply. For the Iphigenie model with 800 K triangles, the time is even over 100 s.

Gilbert et al.'s method performs excellent stability and efficiency. It can give correct results for all the tests as it uses a BSP-tree based Booleans algorithm which allows to explicitly handle all geometric degeneracies without treating a large number of cases. The algorithm also can successfully complete all the Booleans operations in imitating the process of the machining operation. It takes about 26 s for the Iphigenie model with 800 K triangles.

We also compare our method with Campen's method [4] using the Iphigenie model that has 200 K triangles. In [4], the authors evaluated their algorithm using a computer system with Intel Core i7 2.67 GHz CPU and 6GB RAM, and we evaluate our algorithm on a system with Intel Core i5 2.2 GHz CPU and 4GB RAM. Using the same Iphigenie model, Campen's method takes 19.1 s and our method takes only 12.5 s. For the memory utilization, the Campen's method requires about 300M of the computer main memory for the internal representation and our method requires only 160M of the main memory for the whole Boolean algorithm and the two meshes.

In [9, 29], both authors use the Octree technique to accelerate the spatial queries. In [29], the authors combine polygonal and volumetric computations and representations for performing Boolean operations over polygonal meshes. The method is an approximate method. Our method is faster than the method in [29] because the processes of preserving the existing sharp features and reconstructing new features along intersections of the input meshes take much of the time. In [9], the method is well designed and optimized in a multithreaded environment. It is slightly faster than our method when using one thread. We believe that our code can run as fast as the method in [9] when it is optimized.

Our algorithm proposed in this paper provides correct results for all the tests. The algorithm uses an Octree to facilitate the division of the common space of two meshes which can accelerate the speed of intersection detection and reduce memory utilization. GTS uses an AABB tree to divide the whole space of two models. This is the reason that our algorithm is faster than GTS. From the comparisons, it is also faster than Gilbert et al.'s method which does not consider the conversion of meshes. By analyzing floating point arithmetic errors and singularity of intersections, our algorithm shows the ability to

guarantee the unique intersection between a segment and a face, and the continuity of intersections. The algorithm is therefore robust to implement consecutive Boolean operations. It is also a novel technique based on intersected triangles to realize union, intersection and difference operations, which is suitable for both closed and open meshes. The proposed method can create required sub-meshes according to the type of Booleans automatically which also enhances the performance.

4. Conclusions and future work

This work improved the performance and robustness of Boolean operations in the geometric computation. An efficient and robust Booleans method is presented in this paper. The Octree technique is used to facilitate the division of the common space of two meshes in order to accelerate the speed of intersection detections and to reduce memory utilization. Floating point arithmetic errors and singularity of intersections are analyzed to improve the stability of the algorithm. The proposed algorithm can handle very complex meshes. These meshes can be either open or closed because only intersected triangles are used for the point classification. It is fast and robust to meet the requirements of modeling in reverse engineering. Our algorithm uses the Octree technique. The building of the Octree, the intersection detection and the retriangulating intersected triangles can be accelerated in a multithreaded environment after our code has been optimized, which will be done in our future work. Our algorithm can implement consecutive Boolean operations and simulate the process of machining operations. However, with the increasing of the number of Boolean operations, the number of triangles will be increased significantly. This will reduce the efficiency of Booleans operations for the milling simulation. This is also our future work.

Acknowledgements

This study was supported by Aeronautical Science Foundation of China (No.20151652024), Jiangsu Province Science and Technology Support Plan Project (No.BE2014009-3) and Jiangsu Key Laboratory of 3D Printing Equipment and Manufacturing (No.BM2013006).

ORCID

Xiaotong Jiang  <http://orcid.org/0000-0001-9377-239X>
 Qingjin Peng  <http://orcid.org/0000-0002-9664-5326>
 Xiaosheng Cheng  <http://orcid.org/0000-0001-7489-2345>
 Ning Dai  <http://orcid.org/0000-0002-0287-9114>
 Cheng Cheng  <http://orcid.org/0000-0001-5017-4154>
 Dawei Li  <http://orcid.org/0000-0002-9931-0310>

References

- [1] Amato, N. M.; Goodrich, M. T.; Ramos, E. A.: Linear-time Triangulation of a simple Polygon Made easier Via Randomization, *Symposium on Computational Geometry*, 2000, 201–212. <http://dx.doi.org/10.1145/336154.336206>.
- [2] Ayala, D.; Brunet, P.; Juan, R.; Navao, I.: Object representation by means of nonminimal division quadtrees and octrees, *ACM Transactions on Graphics*, 4(1), 1985, 41–59. <http://dx.doi.org/10.1145/3973.3975>.
- [3] Bernstein, G.; Fussell, D.: Fast, Exact, Linear Booleans, *Computer Graphics Forum*, 28(5), 2009, 1269–1278. [http://dx.doi.org/10.1016/0010-4485\(96\)00014-0](http://dx.doi.org/10.1016/0010-4485(96)00014-0).
- [4] Campen, M.; Kobbelt, L.: Exact and robust (self-) intersections for polygonal meshes, *Computer Graphics Forum*, 29(2), 2010, 397–406. <http://dx.doi.org/10.1111/j.1467-8659.2009.01504.x>.
- [5] CGAL, Computational geometry algorithms library, <http://www.cgal.org>, 2011.
- [6] Chen M.; Chen, X. Y.; Tang, K.; Yuen, Matthew M. F.: Efficient Boolean Operation on Manifold Mesh Surfaces, *Computer-Aided Design and Applications*, 7(3), 2010, 405–415. <http://dx.doi.org/10.3722/cadaps.2010.405-415>.
- [7] De, B. M.; Cheong, O.; Van, K.: *Computational geometry: algorithms and applications*, New York Inc: Springer-Verlag, 2008.
- [8] Eitz, M.; Gu, L.: Hierarchical spatial hashing for real-time collision detection, *IEEE international conference on shape modeling and applications*, 2007, 61–70. <http://dx.doi.org/10.1109/SMI.2007.18>.
- [9] Feito, F. R.; Ogayar, C. J.; Segura, R. J.; Rivero, M. L.: Fast and accurate evaluation of regularized Boolean operations on triangulated solids, *Computer-Aided Design*, 45(3), 2013, 705–716. <http://dx.doi.org/10.1016/j.cad.2012.11.004>.
- [10] Frisken, S. F.; Perry, R. N.; Rockwood, A. P.; Jones, T. R.: Adaptively Sampled Distance Fields: A General Representation of Shape for Computer Graphics, *ACM SIGGRAPH*, 2000, 249–254. <http://dx.doi.org/10.1145/344779.344899>.
- [11] Garey, M. R.; Johnson D. S.; Preparata F. P.; Tarjan R. E.: Triangulating a simple polygon, *Information Processing Letters*, 7(4), 1978, 175–179. [http://dx.doi.org/10.1016/0020-0190\(78\)90062-5](http://dx.doi.org/10.1016/0020-0190(78)90062-5).
- [12] GTS, The GNU triangulated surface library, <http://gts.sourceforge.net/>, 2012.
- [13] Guo, K. B.; Zhang, L. C.; Wang, C. J.; Huang, S. H.: Boolean operations of STL moleds based on loop detection, *International Journal of Advanced Manufacturing Technology*, 33, 2007, 627–633. <http://dx.doi.org/10.1007/s00170-006-0876-9>.
- [14] Held, M.: FIST: Fast industrial-strength triangulation of polygons, *Algorithmica*, 30, 2011, 563–596. <http://dx.doi.org/10.1007/s00453-001-0028-4>.
- [15] Hoffmann, C.: Robustness in geometric computations, <http://www.cs.uoi.gr/~fudos/gradmaterial/robust4.pdf>, 2001.
- [16] Hoffmann, C. M.; Hopcroft, John E.; Karasick, M. J.: Robust set operations on polyhedral solids, *IEEE Computer Graphics and Applications*, 9(6), 1989, 50–59. <http://dx.doi.org/10.1109/38.41469>

- [17] Hu, C.-Y.; Patrikalakis, N.-M.; Ye, X.: Robust Interval Solid Modelling Part I: representations, *Computer-Aided Design*, 28(10), 1996, 807–817. [http://dx.doi.org/10.1016/0010-4485\(96\)00013-9](http://dx.doi.org/10.1016/0010-4485(96)00013-9).
- [18] Hu, C.-Y.; Patrikalakis, N.-M.; Ye, X.: Robust Interval Solid Modelling Part II: boundary evaluation, *Computer-Aided Design*, 28(10), 1996, 819–830. [http://dx.doi.org/10.1016/0010-4485\(96\)00014-0](http://dx.doi.org/10.1016/0010-4485(96)00014-0).
- [19] Jacobson, A.; Kavan, L.; Sorkine, H. O.: Robust inside-outside segmentation using generalized winding numbers, *ACM Transactions on Graphics*, 32(4), 2013, 33:1–33:12. <http://dx.doi.org/10.1145/2461912.2461916>.
- [20] Jing, Y. B.; Wang, L. G.; Bi, L.; Chen, J. H.: Boolean Operations on Polygonal Meshes Using OBB Trees, *Proceedings of International Conference on Environmental Science and Information Application Technology, China*, 2009, 619–622. <http://dx.doi.org/10.1109/ESIAT.2009.128>.
- [21] Keyser, J.; Krishnan, S.; Manocha, D.: Efficient and Accurate B-Rep Generation of Low Degree Sculptured Solids Using Exact Arithmetic: I-Representations, *Computer Aided Geometric Design*, 16(9), 1999, 841–859. [http://dx.doi.org/10.1016/S0167-8396\(99\)00032-1](http://dx.doi.org/10.1016/S0167-8396(99)00032-1).
- [22] Kobbelt, L. P.; Botsch, M.; Schwanecke U.; Seidel HP.: Feature sensitive surface extraction from volume data, *ACM SIGGRAPH*, 2001, 57–66. <http://dx.doi.org/10.1145/383259.383265>.
- [23] Kurt M.; Chee Y.: Robust geometric computation (tentative title), <http://cs.nyu.edu/~yap/book/egc>, 2004.
- [24] Laidlaw, D.; Benjamin Trumbore, W.; Hughes, J.: Constructive solid geometry for polyhedral objects, *ACM SIGGRAPH Computer Graphics*, 20(4), 1986, 161–170. <http://dx.doi.org/10.1145/15886.15904>.
- [25] Liu, Y.-J.: Exact Geodesic Metric in 2-Manifold Triangle Meshes Using Edge-based Data Structures, *Computer-Aided Design*, 45(3), 2012, 695–704. <http://dx.doi.org/10.1016/j.cad.2012.11.005>.
- [26] Moller, T.: A Fast Triangle-Triangle Intersection Test, *Journal of Graphics Tools*, 2(2), 1997, 25–30. <http://dx.doi.org/10.1080/10867651.1997.10487472>.
- [27] Museth, K.; Breen, D. E.; Whitaker, R. T.; Barr, A. H.: Level Set Surface Editing Operators, *ACM Transactions on Graphics*, 21(3), 2002, 330–338. <http://dx.doi.org/10.1145/566570.566585>.
- [28] Ogayar, C. J.; Feito, F. R.; Segura, R. J.; Rivero, M. L.: GPU-based evaluation of Boolean operations on triangulated solids, *Ibero-American Symposium in Computer Graphics*, 2006, 121–130. <http://dx.doi.org/10.2312/LocalChapterEvents/siacg/siacg06/121-130>.
- [29] Pavić, D.; Campen, M.; Kobbelt, L.: Hybrid Booleans, *Computer Graphics Forum*, 29(1), 2010, 75–87. <http://dx.doi.org/10.1111/j.1467-8659.2009.01545.x>.
- [30] Requicha, A. A. G.; Voelcker, H. B.: Boolean operations in solid modeling: boundary evaluation and merging algorithms, *Proceedings of the IEEE*, 73(1), 1985, 30–44. <http://dx.doi.org/10.1109/PROC.1985.13108>.
- [31] Seidel R.: A simple and fast incremental randomized algorithm for computing trapezoidal decompositions and for triangulating polygons, *Computational Geometry*, 1(1), 1991, 51–64. [http://dx.doi.org/10.1016/0925-7721\(91\)90012-4](http://dx.doi.org/10.1016/0925-7721(91)90012-4).
- [32] Smith, J. M.; Dodgson, N. A.: A topologically robust algorithm for Boolean operations on polyhedral shapes using approximate arithmetic, *Computer-Aided Design*, 39(2), 2007, 149–163. <http://dx.doi.org/10.1016/j.cad.2006.11.003>.
- [33] Teschner, M.; Heidelberger, B.; Müller, M.; Pomeranets, D.; Gross, M.: Optimized spatial hashing for collision detection of deformable objects, *Proceedings of vision, modeling, visualization VMV*, 2003, 47–54. (DIO: Not Found)
- [34] Toussaint, G.: Efficient triangulation of simple polygons, *The Visual Computer*, 7(5–6), 1991, 280–295. <http://dx.doi.org/10.1007/BF01905693>.
- [35] Tropp, O.; Tal, A.; Shimshoni, I.: A fast triangle to triangle intersection test for collision detection, *Computer Animation and Virtual Worlds*, 17(5), 2006, 527–535. <http://dx.doi.org/10.1002/cav.115>.
- [36] Veblen, O.: Theory on plane curves in non-metrical analysis situs, *Trans Amer Math Soc*, 6(1), 1905, 83–98. <http://dx.doi.org/10.1090/S0002-9947-1905-1500697-4>.
- [37] Wu L.: Poly2Tri: Fast and Robust Simple Polygon Triangulation With/ Without Holes by Sweep Line Algorithm, <http://sites-final.uclouvain.be/mema/Poly2Tri/>, 2005.
- [38] Xu, S. G., Keyser, J.: Fast and robust Booleans on polyhedra, *Computer-Aided Design*, 45: 729–734, 2013. <http://dx.doi.org/10.1016/j.cad.2012.10.036>.